Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI

Thursday, September 21 2006 15:56.06 CDT

Author: igorsk # Views: 168561 Printer Friendly ...

Abstract

Microsoft Visual C++ is the most widely used compiler for Win32 so it is important for the Win32 reverser to be familiar with its inner working. Being able to recognize the compiler-generated glue code helps to quickly concentrate on the actual code written by the programmer. It also helps in recovering the high-level structure of the program.

In part II of this 2-part article (see also: Part I: Exception Handling), I will cover how C++ machinery is implemented in MSVC, including classes layout, virtual functions, RTTI. Familiarity with basic C++ and assembly language is assumed.

Basic Class Layout

To illustrate the following material, let's consider this simple example:

```
class A
{
  int a1;
public:
  virtual int A_virt1();
  virtual int A_virt2();
  static void A_static1();
  void A_simple1();
};

class B
{
  int b1;
  int b2;
public:
  virtual int B_virt1();
  virtual int B_virt2();
};

class C: public A, public B
{
  int c1;
public:
  virtual int A_virt2();
  virtual int B_virt2();
};
```

In most cases MSVC lays out classes in the following order:

- 1. Pointer to virtual functions table (_vtable_ or _vftable_), added only when the class has virtual methods and no suitable table from a base class can be reused.
- 2. Base classes
- 3. Class members

Virtual function tables consist of addresses of virtual methods in the order of their first appearance. Addresses of overloaded functions replace addresses of functions from base classes.

Thus, the layouts for our three classes will look like following:

```
class A size(8):
    +---
 0  | {vfptr}
 4  | a1
    +---

A's vftable:
 0  | &A::A_virt1
 4  | &A::A_virt2

class B size(12):
    +---
 0  | {vfptr}
 4  | b1
 8  | b2
    +---

B's vftable:
 0  | &B::B_virt1
 4  | &B::B_virt2

class C size(24):
    +---
    | +--- (base class A)
 0  | | {vfptr}
 4  | | a1
    | +---
    | +--- (base class B)
 8  | | {vfptr}
12  | | b1
16  | | b2
    | +---
20  | c1
    +---

C's vftable for A:
 0  | &A::A_virt1
 4  | &C::A_virt2

C's vftable for B:
 0  | &B::B_virt1
 4  | &C::B_virt2
```

The above diagram was produced by the VC8 compiler using an undocumented switch. To see the class layouts produced by the compiler, use: -

d1reportSingleClassLayout to see the layout of a single class -d1reportAllClassLayout to see the layouts of all classes (including internal CRT classes) The layouts are dumped to stdout.

As you can see, C has two vftables, since it has inherited two classes which both already had virtual functions. Address of C::A_virt2 replaces address of A::A_virt2 in C's vftable for A, and C::B_virt2 replaces B::B_virt2 in the other table.

Calling Conventions and Class Methods

All class methods in MSVC by default use _thiscall_ convention. Class instance address (_this_ pointer) is passed as a hidden parameter in the ecx register. In the method body the compiler usually tucks it away immediately in some other register (e.g. esi or edi) and/or stack variable. All further adressing of the class members is done through that register and/or variable. However, when implementing COM classes, _stdcall_ convention is used. The following is an overview of the various class method types.

1) **Static Methods**
Static methods do not need a class instance, so they work the same way as common functions. No _this_ pointer is passed to them. Thus it's not possible to reliably distinguish static methods from simple functions. Example:

```
    A::A_static1();
    call    A::A_static1
```

2) **Simple Methods**
Simple methods need a class instance, so _this_ pointer is passed to them as a hidden first parameter, usually using _thiscall_ convention, i.e. in _ecx_ register. When the base object is not situated at the beginning of the derived class, _this_ pointer needs to be adjusted to point to the actual beginning of the base subobject before calling the function. Example:

```
    ;pC->A_simple1(1);
    ;esi = pC
    push    1
    mov ecx, esi
    call    A::A_simple1

    ;pC->B_simple1(2,3);
    ;esi = pC
    lea edi, [esi+8] ;adjust this
    push    3
    push    2
    mov ecx, edi
    call    B::B_simple1
```

As you see, _this_ pointer is adjusted to point to the B subobject before calling B's method.

3) **Virtual Methods**
To call a virtual method the compiler first needs to fetch the function address from the _vftable_ and then call the function at that address same way as a simple method (i.e. passing _this_ pointer as an implicit parameter). Example:

```
    ;pC->A_virt2()
    ;esi = pC
    mov eax, [esi]  ;fetch virtual table pointer
    mov ecx, esi
    call [eax+4]  ;call second virtual method

    ;pC->B_virt1()
    ;edi = pC
    lea edi, [esi+8] ;adjust this pointer
    mov eax, [edi]   ;fetch virtual table pointer
    mov ecx, edi
    call [eax]       ;call first virtual method
```

4) **Constructors and Destructors**
Constructors and destructors work similar to a simple method: they get an implicit _this_ pointer as the first parameter (e.g. ecx in case of _thiscall_ convention). Constructor returns the _this_ pointer in eax, even though formally it has no return value.

RTTI Implementation

RTTI (Run-Time Type Identification) is special compiler-generated information which is used to support C++ operators like dynamic_cast<> and typeid(), and also for C++ exceptions. Due to its nature, RTTI is only required (and generated) for polymorphic classes, i.e. classes with virtual functions.

MSVC compiler puts a pointer to the structure called "Complete Object Locator" just before the vftable. The structure is called so because it allows compiler to find the location of the complete object from a specific vftable pointer (since a class can have several of them). COL looks like following:

```
struct RTTICompleteObjectLocator
{
    DWORD signature; //always zero ?
    DWORD offset;    //offset of this vtable in the complete class
    DWORD cdOffset;  //constructor displacement offset
    struct TypeDescriptor* pTypeDescriptor; //TypeDescriptor of the complete class
    struct RTTIClassHierarchyDescriptor* pClassDescriptor; //describes inheritance hierarchy
};
```

Class Hierarchy Descriptor describes the inheritance hierarchy of the class. It is shared by all COLs for a class.

```
struct RTTIClassHierarchyDescriptor
{
    DWORD signature;      //always zero?
    DWORD attributes;     //bit 0 set = multiple inheritance, bit 1 set = virtual inheritance
    DWORD numBaseClasses; //number of classes in pBaseClassArray
    struct RTTIBaseClassArray* pBaseClassArray;
};
```

Base Class Array describes all base classes together with information which allows compiler to cast the derived class to any of them during execution of the _dynamic_cast_ operator. Each entry (Base Class Descriptor) has the following structure:

```
struct RTTIBaseClassDescriptor
{
    struct TypeDescriptor* pTypeDescriptor; //type descriptor of the class
    DWORD numContainedBases; //number of nested classes following in the Base Class Array
    struct PMD where;        //pointer-to-member displacement info
    DWORD attributes;        //flags, usually 0
};

struct PMD
{
```

```
    int mdisp;  //member displacement
    int pdisp;  //vbtable displacement
    int vdisp;  //displacement inside vbtable
};
```

The PMD structure describes how a base class is placed inside the complete class. In the case of simple inheritance it is situated at a fixed offset from the start of object, and that value is the _mdisp_ field. If it's a virtual base, an additional offset needs to be fetched from the vbtable. Pseudo-code for adjusting _this_ pointer from derived class to a base class looks like the following:

```
    //char* pThis; struct PMD pmd;
    pThis+=pmd.mdisp;
    if (pmd.pdisp!=-1)
    {
      char *vbtable = pThis+pmd.pdisp;
      pThis += *(int*)(vbtable+pmd.vdisp);
    }
```

For example, the RTTI hierarchy for our three classes looks like this:

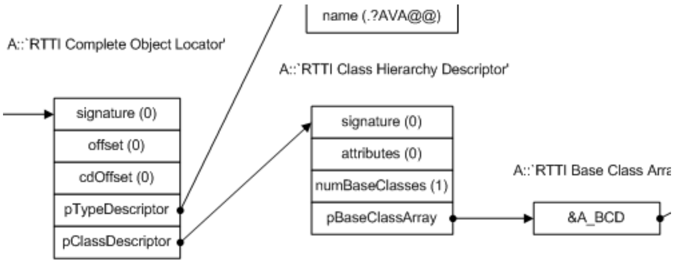


IMAGE EXCERPT (click for full size)
RTTI hierarchy for our example classes

Extracting Information

1) **RTTI**
If present, RTTI is a valuable source of information for reversing. From RTTI it's possible to recover class names, inheritance hierarchy, and in some cases parts of the class layout. My RTTI scanner script shows most of that information. (see Appendix I)

2) **Static and Global Initializers**
Global and static objects need to be initialized before the main program starts. MSVC implements that by generating initializer funclets and putting their addresses in a table, which is processed during CRT startup by the _cinit function. The table usually resides in the beginning of .data section. A typical initializer looks like following:

```
    _init_gA1:
        mov     ecx, offset _gA1
        call    A::A()
        push    offset _term_gA1
        call    _atexit
        pop     ecx
        retn
    _term_gA1:
        mov     ecx, offset _gA1
        call    A::~A()
        retn
```

Thus, from this table way we can find out:

- Global/static objects addresses
- Their constructors
- Their destructors

See also MSVC _#pragma_ directive _init_seg_ [5].

3) **Unwind Funclets**
If any automatic objects are created in a function, VC++ compiler automatically generates exception handling structures which ensure deletion of those objects in case an exception happens. See Part I for a detailed description of C++ exception implementation. A typical unwind funclet destructs an object on the stack:

```
    unwind_1tobase:  ; state 1 -> -1
        lea     ecx, [ebp+a1]
        jmp     A::~A()
```

By finding the opposite state change inside the function body or just the first access to the same stack variable, we can also find the constructor:

```
    lea     ecx, [ebp+a1]
    call    A::A()
    mov     [ebp+__$EHRec$.state], 1
```

For the objects constructed using new() operator, the unwind funclet ensures deletion of allocated memory in case the constructor fails:

```
    unwind_0tobase: ; state 0 -> -1
        mov     eax, [ebp+pA1]
        push    eax
        call    operator delete(void *)
        pop     ecx
        retn
```

In the function body:

```
    ;A* pA1 = new A();
        push
        call    operator new(uint)
        add     esp, 4
        mov     [ebp+pA1], eax
        test    eax, eax
        mov     [ebp+__$EHRec$.state], 0; state 0: memory allocated but object is not yet constructed
```

```
        jz      short @@new_failed
        mov     ecx, eax
        call    A::A()
        mov     esi, eax
        jmp     short @@constructed_ok
    @@new_failed:
        xor     esi, esi
    @@constructed_ok:
        mov     [esp+14h+__$EHRec$.state], -1
     ;state -1: either object was constructed successfully or memory allocation failed
     ;in both cases further memory management is done by the programmer
```

Another type of unwind funclets is used in constructors and destructors. It ensures destruction of the class members in case of exception. In this case the funclets use the _this_ pointer, which is kept in a stack variable:

```
    unwind_2to1:
        mov     ecx, [ebp+_this] ; state 2 -> 1
        add     ecx, 4Ch
        jmp     B1::~B1
```

Here the funclet destructs a class member of type B1 at the offset 4Ch. Thus, from unwind funclets we can find out:

- Stack variables representing C++ objects or pointers to objects allocated with _operator new_.
- Their destructors
- Their constructors
- in case of new'ed objects, their size

4) **Constructors / Destructors Recursion**
This rule is simple: constructors call other constructors (of base classes and member variables) and destructors call other destructors. A typical constructor does the following:

- Call constructors of the base classes.
- Call constructors of complex class members.
- Initialize vfptr(s) if the class has virtual functions
- Execute the constructor body written by the programmer.

Typical destructor works almost in the reverse order:

- Initialize vfptr if the class has virtual functions
- Execute the destructor body written by the programmer.
- Call destructors of complex class members
- Call destructors of base classes

Another distinctive feature of destructors generated by MSVC is that their _state_ variable is usually initialized with the highest value and then gets decremented with each destructed subobject, which make their identification easier. Be aware that simple constructors/destructors are often inlined by MSVC. That's why you can often see the vftable pointer repeatedly reloaded with different pointers in the same function.

5) **Array Construction Destruction**
The MSVC compiler uses a helper function to construct and destroy an array of objects. Consider the following code:

```
    A* pA = new A[n];

    delete [] pA;
```

It is translated into the following pseudocode:

```
    array = new char(sizeof(A)*n+sizeof(int))
    if (array)
    {
      *(int*)array=n; //store array size in the beginning
      'eh vector constructor iterator'(array+sizeof(int),sizeof(A),count,&A::A,&A::~A);
    }
    pA = array;

    'eh vector destructor iterator'(pA,sizeof(A),count,&A::~A);
```

If A has a vftable, a 'vector deleting destructor' is invoked instead when deleting the array:

```
    ;pA->'vector deleting destructor'(3);
    mov ecx, pA
    push 3 ; flags: 0x2=deleting an array, 0x1=free the memory
    call A::'vector deleting destructor'
```

If A's destructor is virtual, it's invoked virtually:

```
    mov ecx, pA
    push 3
    mov eax, [ecx] ;fetch vtable pointer
    call [eax]     ;call deleting destructor
```

Consequently, from the vector constructor/destructor iterator calls we can determine:

- addresses of arrays of objects
- their constructors
- their destructors
- class sizes

6) **Deleting Destructors**
When class has a virtual destructor, compiler generates a helper function - deleting destructor. Its purpose is to make sure that a proper _operator delete_ gets called when destructing a class. Pseudo-code for a deleting destructor looks like following:

```
    virtual void * A::'scalar deleting destructor'(uint flags)
    {
      this->~A();
      if (flags&1) A::operator delete(this);
    };
```

The address of this function is placed into the vftable instead of the destructor's address. This way, if another class overrides the virtual destructor, _operator delete_ of that class will be called. Though in real code _operator delete_ gets overriden quite rarely, so usually you see a call to the default delete(). Sometimes compiler can also generate a vector deleting destructor. Its code looks like this:

```
virtual void * A::'vector deleting destructor'(uint flags)
{
  if (flags&2) //destructing a vector
  {
    array = ((int*)this)-1; //array size is stored just before the this pointer
    count = array[0];
    'eh vector destructor iterator'(this,sizeof(A),count,A::~A);
    if (flags&1) A::operator delete(array);
  }
  else {
    this->~A();
    if (flags&1) A::operator delete(this);
  }
};
```

I skipped most of the details on implementation of classes with virtual bases since they complicate things quite a bit and are rather rare in the real world. Please refer to the article by Jan Gray[1]. It's very detailed, if a bit heavy on Hungarian notation. The article [2] describes an example of the virtual inheritance implementation in MSVC. See also some of the MS patents [3] for more details.

Appendix I: ms_rtti4.idc

This is a script I wrote for parsing RTTI and vftables. You can download the scripts associated with both this article and the previous article from Microsoft VC++ Reversing Helpers. The script features:

- Parses RTTI structures and renames vftables to use the corresponding class names.
- For some simple cases, identifies and renames constructors and destructors.
- Outputs a file with the list of all vftables with referencing functions and class hierarchy.

Usage: after the initial analysis finishes, load ms_rtti4.idc. It will ask if you want to scan the exe for the vtables. Be aware that it can be a lengthy process. Even if you skip the scanning, you can still parse vtables manually. If you do choose to scan, the script will try to identify all vtables with RTII, rename them, and identify and rename constructors and destructors. In some cases it will fail, especially with virtual inheritance. After scanning, it will open the text file with results.

After the script is loaded, you can use the following hotkeys to parse some of the MSVC structures manually:

- Alt-F8 - parse a vtable. The cursor should be at the beginning of the vtable. If there is RTTI, the script will use the class name from it. If there is none, you can enter the class name manually and the script will rename the vtable. If there is a virtual destructor which it can identify, the script will rename it too.
- Alt-F7 - parse FuncInfo. FuncInfo is the structure present in functions which have objects allocated on the stack or use exception handling. Its address is passed to _CxxFrameHandler in the function's exception handler:

```
mov eax, offset FuncInfo1
jmp _CxxFrameHandler
```

  In most cases it is identified and parsed automatically by IDA, but my script provides more information. You can also use ms_ehseh.idc from the first part of this article to parse all FuncInfos in the file.

  Use the hotkey with cursor placed on the start of the FuncInfo structure.
- Alt-F9 - parse throw info. Throw info is a helper structure used by _CxxThrowException to implement the _throw_ operator. Its address is the second argument to _CxxThrowException:

```
lea     ecx, [ebp+e]
call    E::E()
push    offset ThrowInfo_E
lea     eax, [ebp+e]
push    eax
call    _CxxThrowException
```

Use the hotkey with the cursor placed on the start of the throw info structure. The script will parse the structure and add a repeatable comment with the name of the thrown class. It will also identify and rename the exception's destructor and copy constructor.

Appendix II: Practical Recovery of a Class Structure

Our subject will be MSN Messenger 7.5 (msnmsgr.exe version 7.5.324.0, size 7094272). It makes heavy use of C++ and has plenty of RTTI for our purposes. Let's consider two vftables, at .0040EFD8 and .0040EFE0. The complete RTTI structures hierarchy for them looks like following:

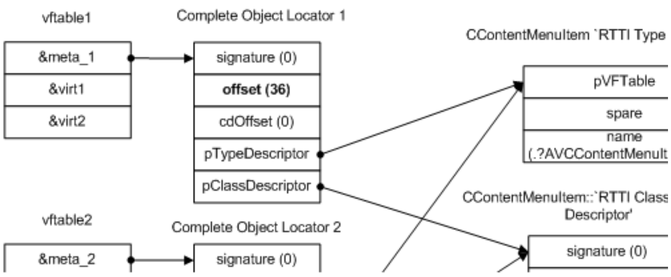
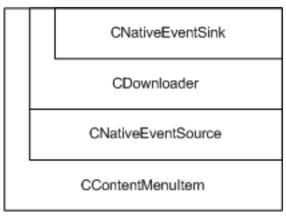


IMAGE EXCERPT (click for full size)

RTTI hierarchy for MSN Messenger 7.5

So, these two vftables both belong to one class - CContentMenuItem. By checking its Base Class Descriptors we can see that:

- CContentMenuItem contains three bases that follow it in the array - i.e. CDownloader, CNativeEventSink and CNativeEventSource.
- CDownloader contains one base - CNativeEventSink.
- Hence, CContentMenuItem inherits directly from CDownloader and CNativeEventSource, and CDownloader in turn inherits from CNativeEventSink.
- CDownloader is situated in the beginning of the complete object, and CNativeEventSource is at the offset 0x24.



So we can conclude that the first vftable lists methods of CNativeEventSource and the second one of either CDownloader or CNativeEventSink (if neither of them had virtual methods, CContentMenuItem would reuse the vftable of CNativeEventSource). Now let's check what refers to these tables. They both are referred by two functions, at .052B5E0 and .052B547. (That reinforces the fact that they both belong to one class.) Moreover, if we look at the beginning of the function at .052B547, we see the _state_ variable initialized with 6, which means that that function is the destructor. As a class can have only one destructor, we can conclude that .052B5E0 is its constructor. Let's looks closer at it:

```
CContentMenuItem::CContentMenuItem   proc near
this = esi
    push    this
    push    edi
    mov     this, ecx
    call    sub_4CA77A
    lea     edi, [this+24h]
    mov     ecx, edi
    call    sub_4CBFDB
    or      dword ptr [this+48h], 0FFFFFFFFh
    lea     ecx, [this+4Ch]
    mov     dword ptr [this], offset const CContentMenuItem::'vftable'{for 'CContentMenuItem'}
    mov     dword ptr [edi], offset const CContentMenuItem::'vftable'{for 'CNativeEventSource'}
    call    sub_4D8000
    lea     ecx, [this+50h]
    call    sub_4D8000
    lea     ecx, [this+54h]
    call    sub_4D8000
    lea     ecx, [this+58h]
    call    sub_4D8000
    lea     ecx, [this+5Ch]
    call    sub_4D8000
    xor     eax, eax
    mov     [this+64h], eax
    mov     [this+68h], eax
    mov     [this+6Ch], eax
    pop     edi
    mov     dword ptr [this+60h], offset const CEventSinkList::'vftable'
    mov     eax, this
    pop     this
    retn
sub_52B5E0      endp
```

The first thing compiler does after prolog is copying _this_ pointer from ecx to esi, so all further addressing is done based on esi. Before initializing vfptrs it calls two other functions; those must be constructors of the base classes - in our case CDownloader and CNativeEventSource. We can confirm that by going inside each of the functions - first one initializes its vfptr field with CDownloader::'vftable' and the second with CNativeEventSource::'vftable'. We can also investigate CDownloader's constructor further - it calls constructor of its base class, CNativeEventSink.

Also, the _this_ pointer passed to the second function is taken from edi, which points to this+24h. According to our class structure diagram it's the location of the CNativeEventSource subobject. This is another confirmation that the second function being called is the constructor of CNativeEventSource.

After calling base constructors, the vfptrs of the base objects are overwritten with CContentMenuItem's implementations - which means that CContentMenuItem overrides some of the virtual methods of the base classes (or adds its own). (If needed, we can compare the tables and check which pointers have been changed or added - those will be new implementations by CContentMenuItem.)

Next we see several function calls to .04D8000 with _ecx_ set to this+4Ch to this+5Ch - apparently some member variables are initialized. How can we know whether that function is a compiler-generated constructor call or an initializer function written by the programmer? There are several hints that it's a constructor.

- The function uses _thiscall_ convention and it is the first time these fields are accessed.
- The fields are initialized in the order of increasing addresses.

To be sure we can also check the unwind funclets in the destructor - there we can see the compiler-generated destructor calls for these member variables.

This new class doesn't have virtual methods and thus no RTTI, so we don't know its real name. Let's name it RefCountedPtr. As we have already determined, 4D8000 is its constructor. The destructor we can find out from the CContentMenuItem destructor's unwind funclets - it's at 63CCB4.

Going back to the CContentMenuItem constructor, we see three fields initialized with 0 and one with a vftable pointer. This looks like an inlined constructor for a member variable (not a base class, since a base class would be present in the inheritance tree). From the used vftable's RTTI we can see that it's an instance of CEventSinkList template.

Now we can write a possible declaration for our class.

```
class CContentMenuItem: public CDownloader, public CNativeEventSource
{
/* 00 CDownloader */
/* 24 CNativeEventSource */
/* 48 */ DWORD m_unknown48;
/* 4C */ RefCountedPtr m_ptr4C;
/* 50 */ RefCountedPtr m_ptr50;
/* 54 */ RefCountedPtr m_ptr54;
/* 58 */ RefCountedPtr m_ptr58;
/* 5C */ RefCountedPtr m_ptr5C;
/* 60 */ CEventSinkList m_EventSinkList;
/* size = 70? */
};
```

We can't know for sure that the field at offset 48 is not a part of CNativeEventSource; but since it wasn't accessed in CNativeEventSource constructor, it is most probably a part of CContentMenuItem. The constructor listing with renamed methods and class structure applied:

```
public: __thiscall CContentMenuItem::CContentMenuItem(void) proc near
    push    this
    push    edi
    mov     this, ecx
    call    CDownloader::CDownloader(void)
    lea     edi, [this+CContentMenuItem._CNativeEventSource]
    mov     ecx, edi
    call    CNativeEventSource::CNativeEventSource(void)
    or      [this+CContentMenuItem.m_unknown48], -1
    lea     ecx, [this+CContentMenuItem.m_ptr4C]
    mov     [this+CContentMenuItem._CDownloader._vfptr], offset const CContentMenuItem::'vftable'{for 'CContentMenuItem'}
    mov     [edi+CNativeEventSource._vfptr], offset const CContentMenuItem::'vftable'{for 'CNativeEventSource'}
    call    RefCountedPtr::RefCountedPtr(void)
    lea     ecx, [this+CContentMenuItem.m_ptr50]
    call    RefCountedPtr::RefCountedPtr(void)
    lea     ecx, [this+CContentMenuItem.m_ptr54]
    call    RefCountedPtr::RefCountedPtr(void)
    lea     ecx, [this+CContentMenuItem.m_ptr58]
    call    RefCountedPtr::RefCountedPtr(void)
    lea     ecx, [this+CContentMenuItem.m_ptr5C]
    call    RefCountedPtr::RefCountedPtr(void)
    xor     eax, eax
    mov     [this+CContentMenuItem.m_EventSinkList.field_4], eax
    mov     [this+CContentMenuItem.m_EventSinkList.field_8], eax
```

```
    mov     [this+CContentMenuItem.m_EventSinkList.field_C], eax
    pop     edi
    mov     [this+CContentMenuItem.m_EventSinkList._vfptr], offset const CEventSinkList::'vftable'
    mov     eax, this
    pop     this
    retn
public: __thiscall CContentMenuItem::CContentMenuItem(void) endp
```

Links and References

[1] http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvc/html/jangrayhood.asp
with illustrations (but in Japanese): http://www.microsoft.com/japan/msdn/vs_previous/visualc/techmat/feature/jangrayhood/
C++: Under the Hood (PDF)

[2] http://www.lrdev.com/lr/c/virtual.html

[3] Microsoft patents which describe various parts of their C++ implementation. Very insightful.

- 5410705: Method for generating an object data structure layout for a class in a compiler for an object-oriented programming language
- 5617569: Method for implementing pointers to members in a compiler for an object-oriented programming language
- 5754862: http://freepatentsonline.com/5854931.html Method and system for accessing virtual base classes
- 5297284: Method and system for implementing virtual functions and virtual base classes and setting a this pointer for an object-oriented programming language
- 5371891: Method for object construction in a compiler for an object-oriented programming language
- 5603030: Method and system for destruction of objects using multiple destructor functions in an object-oriented computer system
- 6138269: Determining the actual class of an object at run time

[4] Built-in types for compiler's RTTI and exception support.
http://members.ozemail.com.au/~geoffch@ozemail.com.au/samples/programming/msvc/language/predefined/index.html

[5] #pragma init_seg
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/_predir_init_seg.asp

Article Comments

Write Comment / View Complete Comments

Username	Comment Excerpt	Date
cl001	[b][url=http://lululemonsales.webs.com/]Lululem...	Monday, May 6 2013 02:13.31 CDT
julyDragon919	Hi! you ve just made me smile! i was having ...	Tuesday, August 7 2012 07:27.30 CDT
Shine	good article£¬by what method do you trace it?	Thursday, August 4 2011 21:23.55 CDT
martinkro	great artical ,thank you!!	Tuesday, July 19 2011 06:58.50 CDT
EliteKnites	i have some doubts on this.. typeid is returnin...	Tuesday, May 10 2011 02:06.48 CDT
EliteKnites	Thank you so much.. this paper gives a clear cu...	Tuesday, May 10 2011 02:03.28 CDT
qxsl2000	it seems like c++ object hierarchy to be decomp...	Wednesday, March 30 2011 03:47.06 CDT
roczhang	Great paper. I have took almost two days to wan...	Thursday, March 3 2011 13:47.08 CST
tcljg2008	very very good!	Saturday, December 18 2010 05:00.26 CST
hwwh1999	Mark and study	Saturday, September 18 2010 10:04.12 CDT
FloydTammie31	Houses are quite expensive and not everyone can...	Sunday, September 12 2010 04:35.28 CDT
Externalist	I've also read this some time ago but never rea...	Thursday, January 29 2009 20:58.09 CST
Sirmabus	[url=http://www.openrce.org/blog/view/1344/Clas...	Thursday, January 22 2009 04:26.19 CST
Sirmabus	Thanks the vtable finder/namer script functiona...	Wednesday, December 24 2008 00:45.52 CST
dnix	wonder whether these structures found by these ...	Tuesday, August 19 2008 03:32.23 CDT
igorsk	Well, I was disassembling c1xx to check how it ...	Friday, September 22 2006 16:30.48 CDT
MohammadHosein	actually didnt read the whole article yet , but...	Friday, September 22 2006 15:23.54 CDT
linestyle	great work!!:)	Thursday, September 21 2006 20:02.11 CDT